Compiling for Itanium™ Architecture: Triumphs and Challenges

Wei Li

Principal Engineer and Manager

Itanium Compiler Development

Intel® Compiler Lab

wei.li@intel.com

Dec 2, 2001

Overview of This Presentation

- Many good compiler technologies and research results have been developed for EPIC so far. This talk is not intended to be a survey.
- Agenda
  - Part I: Current successes with *product compilers*
    - Brief summary on two categories of program structures: loops and acyclic code
  - Part II: EPIC compiler challenges faced in the real world
    - Main portion of this talk
    - Hope to see new compiler technologies developed for EPIC

Background

- Itanium™ architecture, an EPIC architecture
  - Many generations of Itanium architecture in the product pipeline
  - First generation came out this year.
  - Second generation (code named McKinley) coming next year
- Compilers
  - Product compilers for Itanium architecture in various corporations
  - Research EPIC compilers in many research labs and universities.
- Grow architecture and compiler together
  - This is only the beginning of EPIC.
  - Both architecture and compiler will improve overtime
    - Improved architecture implementations enable more effective compiler optimizations and scheduling.
    - Improved compilers realize the full architecture potential.

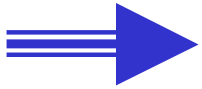
What works well: loops

- Effective software pipelining (e.g. modulo scheduling)
  - Both counted and while loops
  - Predication to eliminate control flow
  - Control and data speculation
- Extensive set of loop optimizations
  - Cache optimizations
  - Using large register set
  - Rotating registers
- Very effective data prefetching
  - Making use of rotating registers
- Leading performance achieved

Example: DAXPY

- Original code: 2 FP ops, 2 loads, 1 store/1 iteration
- Compiled: 1 fma, 1 ldp, 1 st's, 0.25 lfetches/1 iteration
- (nearly) optimal (9 cycle schedule/8 iterations)

```
for i = 1, 1000
  y[i] = y[i] + a*x[i]
end for
```



```
.b1_5:
{  .mmi
 (p16)      ldfpd          f43,f40=[r47]   //0: 13   74
 (p16)      ldfpd          f37,f34=[r49]   //0: 13   86
            nop.i          0 ;;

} ... many bundles with load, fma sequence
 {  .mfi
 (p18)      stfd           [r14]=f67,64    //22: 13   84
 (p17)      fma.d          f66=f7,f49,f55  //13: 13   83
 (p16)      add            r46=64,r47      //4: 13  107 B7
} ... few more stores
{  .mmi
 (p18)      stfd           [r44]=f47       //25: 13   76
 (p16)      lfetch.nt1     [r9],64         //7: 13   97
 (p16)      add            r35=64,r36 ;;   //7: 13  100 B7
}
{  .mfb
 (p18)      stfd           [r3]=f33,64     //26: 13   78
            nop.f          0
            br.ctop.sptk   .b1_5 ;;        //8: 13  113 B7
```

What works well: acyclic code (1)

- Compiler technologies designed to exploit ILP
  - Wealth of techniques available on scheduling, control and data speculation, predication etc
- Scheduler
  - Acyclic regions with single entry/multiple exit
  - Nested regions
  - Cycle-based schedulers
  - Simple code size awareness
- Control and data speculation
  - Integrated naturally into the scheduler
  - Machinery to generate recovery code
  - Simple heuristics

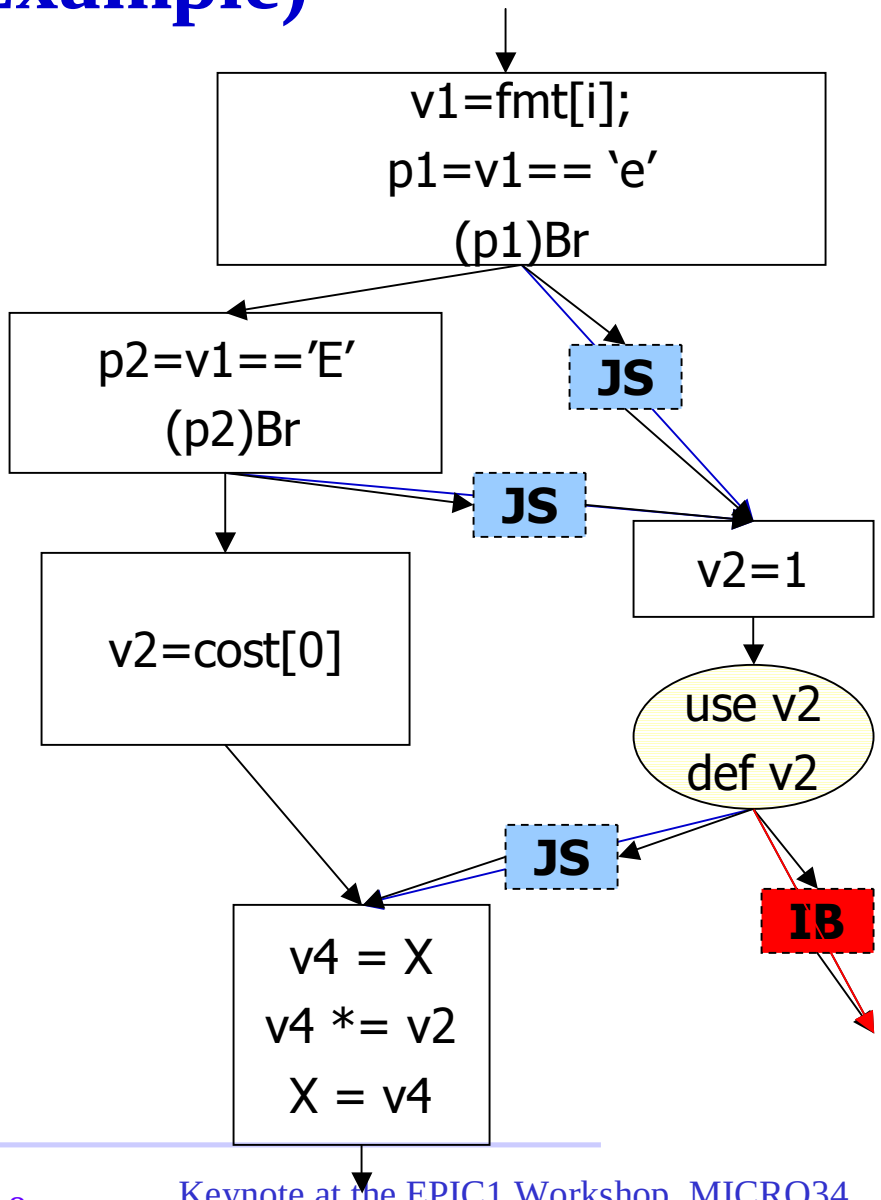
What works well: acyclic code (2)

- Predication
  - If-conversion to eliminate branches
  - Predicate optimizations
  - Predicate aware register allocators

- Profile exploited for better performance
  - Influenced many components of the compiler, such as the scheduler, inliner etc
  - Delivers excellent performance improvement for server applications

Scheduling Region (Example)

```
if (fmt[i]== 'e' || fmt[i]=='E') {
    total = 1;
    for (j=0; j<cnt[i];j++) {
        total *= cost[j];
        if (cost[j] == 0)
            return;
    }
}
else
    total = cost[0];
X *= total
```



So any challenges left?

- Yes.
- Now, let’s look at some performance data from a database server

Sample Server Performance Data on first generation Itanium™ processor

CPI Cycle Breakdown	CPI
Backend Pipeline Flush Cycles	10%
Data Access Cycles	**32%**
Scoreboard Dependency Cycles	2%
RSE Active Cycles	7%
Issue Limit Cycles	11%
Instruction Access Cycles	**31%**
Taken Branch Cycles	2%
Unstalled Pipeline Cycles	6%
Total	100%

Beyond ILP

- Data and instruction accesses are the dominating issues for now.
  - Cache misses
  - TLB misses
- Any ILP improvement will not be visible, until the above problems are solved.

Top 10 EPIC Compiler Challenges

1. Managing data caches/DTLB for acyclic code
2. Managing instruction cache/ITLB
3. More effective use of control speculation
4. More effective use of data speculation
5. Creative use of predication
6. Performance monitor feedback
7. Software pipelining extensions
8. Profiling
9. Whole program optimizations
10. Performance analysis tools

We also have many EPIC Features!

1. Control speculation
2. Data speculation
3. Data prefetching
4. Predication
5. Rotating registers
6. RSE
7. Cache hints
8. Instruction prefetch
9. Software pipelining support
10. Performance monitors

Let’s push the compiler technology boundaries for EPIC!

1. Managing Data Caches/DTLB

- Acyclic codes, function calls.
- Loops for linked lists, Hash-tables, sparse matrix
- Reducing or tolerating Dcache misses
  - Data prefetching. Address computation too close to loads. Distance too small to cover memory latency. Predict future address.
  - Scheduling for cache misses
    - e.g. increase distance between load and use
  - Control speculation
  - Data speculation
  - Identifying missing accesses
- Reducing DTLB misses
  - Data layout optimizations such as structure layout and splitting.

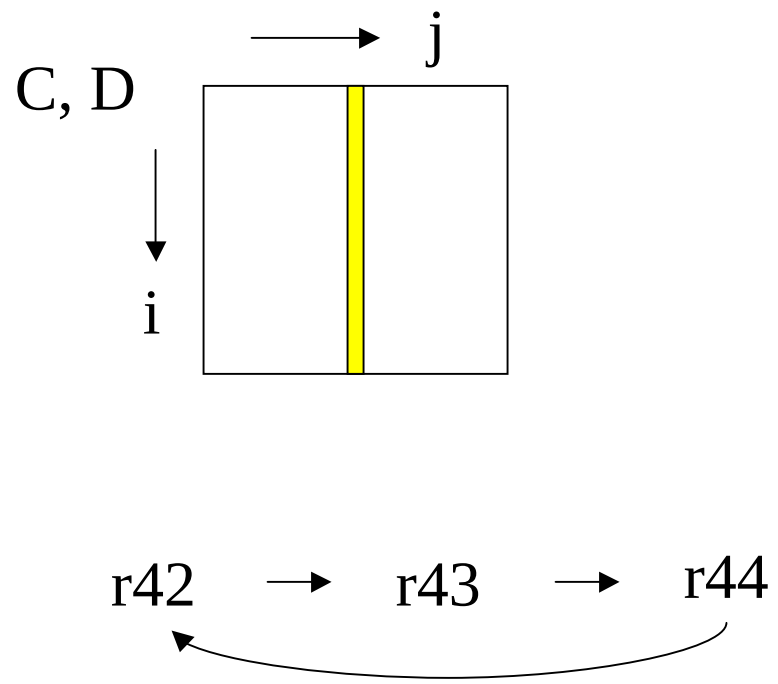
Example: scalar code

- Small distance between the function call to check() and the load of *p_last in the loop.
- Not effective to prefetch above check().

```
void foo (ztype *status)
{
   int last = *p_last;                    // cache hit for *p_last
   if (check (n, i))                      // *p_last evicted
     *status = Err;
   // high cache miss rate for *p_last
   for (n = *p_last; n>last; n--)         // average trip count 1
     …
}
```

Any Creative Use of EPIC Features?

```
for j = 1, n
  r43 = address of D(2+k,j)
  r44 = address of C(1+k,j)
  for i = 1, m
     prefetch [r44]
     r42 = r44+16
     r44 = r43
     r43 = r42
     C(i,j) = D(i-1,j) + D(i+1,j)
  end_for
end_for
```



Uses just one prefetch instruction for multiple arrays. Achieved by using rotating registers, but this is not acyclic code.

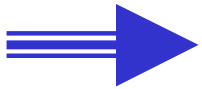
2. Managing Instruction Cache/TLB

- Icache/ITLB misses for large applications
- Reducing Icache/ITLB misses
  - Function layout (page faults, cache associativity)
  - Function splitting
  - Basic block ordering
  - Scheduling for code size
  - Instruction prefetching
    - Streaming prefetch
    - Hint prefetch

Scheduling for Code Size

- Utilize slack for code size.
  - Arrange instructions to better utilize bundles
- Tradeoff between ILP and code size:
  - E.g. compensation code.

```
// 3 bundles
0:    f32 = f34 + f35, I1, I2;;
1:    r32 = r33 + r34, nop, nop;
5:    st [r40] = f32, nop, nop
```



```
// 2 bundles
0:    f32 = f34 + f35, I1, I2;
5:    st [r40]=f32, r32=r33+r34, nop
```

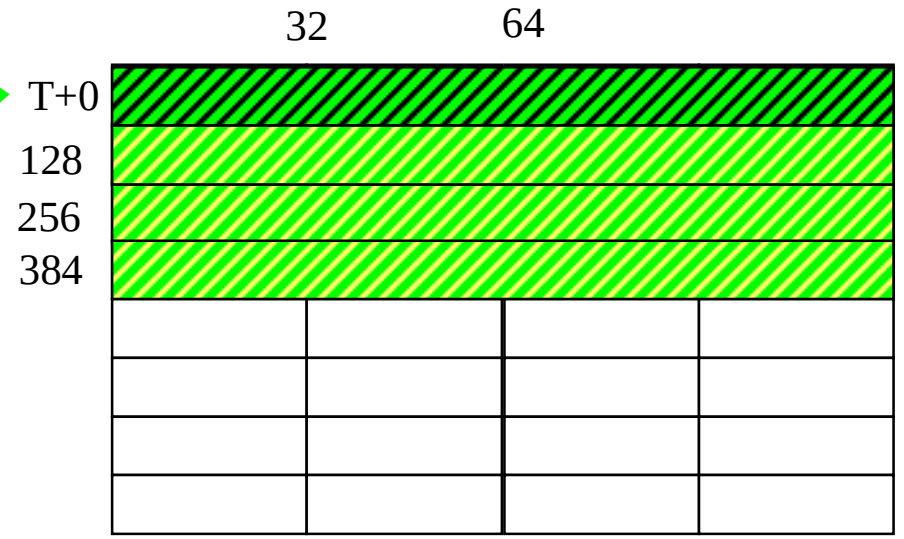
Instruction Prefetching

- Streaming and hint prefetch.

```
// hint
brp.many/few

    ..cycles…

// streaming
(px) br.many T
```



Issues:

- When to use streaming? What if the branch is not taken?
- Is brp worth a new bundle? .few vs .many?

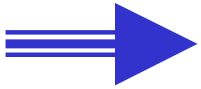
3. Cost of Control Speculation

- Understanding the cost of speculation on the right path
  - Code size (like chk.s and recovery code)
  - Register pressure
- Understand the cost of speculation on the wrong path
  - TLB lookup (DTC, DTLB and VHPT)
  - Cache pollution
  - Avoid cache miss penalty due to speculating uses
  - Code size
- How do we get the benefit and avoid the cost?
  - E.g. Utilizing slack to avoid unnecessary speculation?

Example: cost of speculation

- The branch is likely not taken. Speculation helps.
- But cache miss latency exposed, if the branch is taken.

```
      (p1) br els;;
      r1 = ld   // prob .7
         = r1
      ....
els:              // prob .3
      ....
nxt:
```



```
      r1 = ld.s
//cache miss!
         = r1

(p1) br els;;
      chk.s
      ....
els:
      ....
nxt:
```

4. Data Speculation

- More opportunities in real application environment
  - without whole-program alias analysis
  - However, one has to assume a best possible disambiguator.
- When to speculate?
  - Determine low-probability aliases
- Modeling the cost of advanced load
  - E.g. code size
- Scheduling with data speculation
  - Utilizing slack?
- Optimizations with data speculation
  - E.g. speculative loop invariant removal

Example: utilize slack

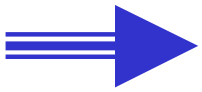
- No advantage to speculate ld [r45], even it is likely independent of st [r4].

```
st [r4]= r3;;
r6 = ld [r44]          // likely dependent with st
r7 = ld [r45];;        // likely independent with st
   = add r6, r7
```

5. Creative Use of Predication

- E.g. partial predication
  - Eliminate branches.
  - Better code bundling
  - Hot short path not hurt by the cold long path

```
      (pF) br els;;
      i1, nop, br nxt;; //prob 0.95
els: i2, nop, nop;;    //prob 0.05
      i3, i4, i5;;
nxt:
```



```
      (pT) i1, (pF) i2, (pT) br nxt;;
      i3, i4, i5;;
nxt:
```

6. Performance Monitor Feedback

- Current profile info:
  - Branch probability
  - Basic block count
  - Value profiling
- Performance counters (low-cost)
  - Cache miss
  - TLB miss
  - Branch probability
  - Branch misprediction rate
- Design heuristics. How to use this effectively?
- Profiling user model

7. Software Pipelining

- Low trip count innermost loop.
- Creative use of predication and data speculation?

```
for (r = 0; r < rows; r++) {
   float s = I[r];
   for (e = row(r); e < row(r+1); e++) {
      s += M[e] * V[C[e]];
   }
   S[r] = s/D[r];
}
```

- low trip count inner loop => swp overhead not amortized
- loop bounds data dependent => complete unroll not possible
- outer loop trip counts large => inefficiency exaggerated
- accessed indirectly => cache latency exposed, prefetch hard

8. Profiling

- Profile-driven optimizations deliver large performance gain for EPIC.
- Profile-driven optimizations under small source changes
  - Incremental changes to profile?
  - Incremental profile collection?
- Some applications have no profile
  - E.g. shipped as a library. Different workloads for users.
  - E.g. no representative profile
  - Optimizations work well with and without profile
  - Re-optimization of binaries with profile?

9. Whole Program Analysis

- Peak performance often reached via detecting whole program.
- Whole program for some applications not possible at compile time.
  - E.g. shipped as a library.
- Better way to achieve the effect of whole program
  - E.g. summary information on individual modules.

10. Performance Analysis Tools

- Current tools:
  - hot spots, but what if the profile is flat?
  - IP-based performance info like cache miss rate. Local info. Scope too small to drive optimizations.
  - Aggregated performance data. Not enough to guide the design of new optimizations.
- How far are we from the optimal?
  - E.g. for loops, we know the min II.
  - Theoretical models (like integer programming) for scheduling
  - Data locality
  - Code locality (e.g. block ordering)

Other Opportunities?

- Exploiting multithreading
  - Optimizing multithreaded applications using IPF features
  - Exploiting multi-threaded IPF architecture
    - Speculative pre-computation
- Exploiting EPIC for JIT
- Dynamic optimizations
- Better debugging support for optimized code

Summary

- Early successes with compiling for EPIC.
  - Effective loop scheduling and data locality techniques. Excellent performance results.
  - Scheduling and optimizations of acyclic code for ILP
- Data and instruction accesses in acyclic code are the main challenges for now.
  - Any ILP improvement will likely not be visible.